Breadth first search on multiple GPUs

M. Bernaschi\textsuperscript{1}, G. Carbone\textsuperscript{2}, M. Fatica\textsuperscript{4}, E. Mastrostefano\textsuperscript{2}, D. Rossetti\textsuperscript{3}

\textsuperscript{1}National Research Council of Italy \\
\textsuperscript{2}Sapienza University Rome-Italy \\
\textsuperscript{3}National Institute of Nuclear Physics-Italy \\
\textsuperscript{4}Nvidia U.S.

March 19, 2013
Large Networks and Graphs

- Large scale networks are often represented as graphs with hundreds million (even billions) of edges
- Real-world networks often feature a power-law degree distribution

<table>
<thead>
<tr>
<th>Connections</th>
<th>% Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10</td>
<td>67</td>
</tr>
<tr>
<td>10-100</td>
<td>23</td>
</tr>
<tr>
<td>100-1000</td>
<td>8</td>
</tr>
<tr>
<td>1000-10000</td>
<td>1</td>
</tr>
<tr>
<td>10000-100000</td>
<td>$8 \times 10^{-2}$</td>
</tr>
<tr>
<td>100000-6000000</td>
<td>$1.5 \times 10^{-4}$</td>
</tr>
</tbody>
</table>
Breadth First Search
High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns
- How do GPU perform running such algorithms?
  - GPU main memory is still limited to 6 GB.
  - For large datasets, cluster of GPUs are required.
High performance graph algorithms

- Most of graph algorithms have low arithmetic intensity and irregular memory access patterns.
- How do GPU perform running such algorithms?
  - GPU main memory is still limited to 6 GB.
  - For large datasets, cluster of GPUs are required.
- Several graph-theoretical challenges: DIMACS9, SCA#2, Graph500.
Overview

- Distributed **Breadth First Search (BFS)**
- Developed according to the Graph 500 specifications:
  - Performance metric: Traversed Edges Per Second (TEPS)
  - graph has *real-world* properties (RMAT generator)
  - vertices are encoded as 64-bit integers.
- Implementation for GPU clusters
  - Hybrid Programming paradigm: **CUDA + Message Passing** (MPI and APEnet).

![Diagram showing BFS on GPU clusters]

G500 Steps

- Generator
- Build data structure
- BFS
- parent array
- TEPS

GPU1 → GPU2 → ... → GPU{n}

MPI
Serial BFS

$i^{th}$ Iteration

Q_{BFS} \hspace{1cm} U_3 \hspace{0.5cm} U_7 \hspace{0.5cm} U_{21} \hspace{0.5cm} U_{16} \hspace{0.5cm} \ldots \hspace{0.5cm} U_i \hspace{0.5cm} U_j

Dequeue(Q_{BFS}, U_3) \hspace{1cm} U_3

Visit neighbors \hspace{1cm} V_7 \hspace{0.5cm} V_3 \hspace{0.5cm} \ldots

Filter vertices \hspace{1cm} V_7 \hspace{0.5cm} \times \hspace{0.5cm} \ldots

parents/distance update

Enqueue vertices \hspace{1cm} \ldots \hspace{1cm} U_i \hspace{0.5cm} U_j \hspace{0.5cm} V_7 \hspace{0.5cm} \ldots

Output

vertices labels \hspace{1cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 3 \hspace{0.5cm} 4 \hspace{0.5cm} 5 \hspace{0.5cm} 6 \hspace{0.5cm} 7 \hspace{0.5cm} 8

parent array \hspace{1cm} 1 \hspace{0.5cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 1 \hspace{0.5cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 6 \hspace{0.5cm} 4

distance array \hspace{1cm} 0 \hspace{0.5cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 1 \hspace{0.5cm} 1 \hspace{0.5cm} 2 \hspace{0.5cm} 3 \hspace{0.5cm} 2
Parallel BFS

**i^{th} Iteration**

- **Current Queue (CQ):**
  - $U_3, U_7, U_{21}, U_{16}, \ldots, U_i, U_j$

- **Next Level Frontier Set (NLFS):**
  - Vectors from $0, \ldots, L_2$
  - Vectors from $0, \ldots, \ldots, L_1$

- **Drop already seen vertices**
- **Update parents**
- **Parallel Enqueue**

**EXPAND**

**CONTRACT**

- **Parallel vs. Serial:**
  - two queues: Current Queue and Next Queue
  - Next Level Frontier Set (NLFS):
    - the set of vertices to be visited in the next level
  - the enqueue operation is critical.
Distributed data structure

Edge list

- Edge list with: \( |V| = 2^{SCALE} \), \( |M| = 16 \times 2^{SCALE} \)
- Each task generates a subset of the edges list in the form: \((U_0, V_0), (U_1, V_1), ...\)
- Currently, edges are assigned to tasks via a simple rule:
  - edge \((U_i, V_j) \in T_k\) if \(U_i \mod \#T = k\)
  - if \(\#T = 16\), \(U = 200 \in\) task 8.

Compressed Sparse Row (CSR) data structure

- CSR is simple and has minimal memory requirements

![CSR Diagram](image-url)
Straightforward BFS on a cluster of GPUs

- each vertex $U_i$ of $Q_{BFS}$ is assigned to one thread $t_i$
- each thread $t_i$ visits all the neighbors $V_j$ of its vertex ($j \in 0, \ldots, d - 1$ where $d$ is the degree of $U$)
- if $V_j$ is local: visit it
  - update the parent array
  - enqueue $V_j$
- if $V_j$ is not local: send to its owner
- receive vertices $V_k$ from other tasks
  - update the parent array
  - enqueue $V_k$

If $Q_{BFS} == 0$ terminate. Output parent array and TEPS
Straightforward Multi-GPU BFS: Results

New computational and communication parts:

- we don’t know in advance how many vertices of the NLFS are local/non-local
- build the array to send
- communication among nodes (involves a passage through the hosting CPU)
- enqueue received data.

Weak scaling plot

(infiniband cluster @ Cineca, 32 nodes: 2xM2070, 2x six-core intel Westmere)

On 1 GPU $2^{21}$ vertices, on 64 GPUs $2^{27}$ vertices

Poor TEPS scaling
Straightforward BFS: issues

GPU-related issues:

- threads workloads are unbalanced. Remember the power law distribution with few vertices having a huge number of edges
- memory access patterns can be irregular
- algorithm relies on the use of **Atomic** operations.

Communication-related issues:

- multiple copies of the same vertex are sent.

![Diagram showing BFS on GPU clusters with processors 1, 2, and k, and vertices V0, V1, V2, V3, V5, V7, V9, V21. The neighbors of U3, U7, and U2 are also indicated.](image)
A better BFS on multiple GPUs

Algorithm carries out four steps at each iteration:

1. computes the total number of neighbors, say \( m \)
2. with \( m \) threads gathers the sets of neighbors (NLFS) in a contiguous array
3. with \( m \) threads prunes the contiguous array of NLFS by means of Sort-Unique
4. exchanges vertices with other tasks.

The NLFS is a set of non-contiguous lists of neighbors stored in the global data structure that represents the graph.
Sort-Unique BFS

Step 1: build the new offset array and compute the total number of neighbors

- start $k$ threads, one for each element of $Q_{BFS}$

- build $Q_{\text{deg}}$, by substituting each vertex with its degree

- perform a prefix-sum operation on $Q_{\text{deg}}$ to build the New Offset array (by using the Thrust library)
Sort-Unique BFS

Step 1: build the new offset array and compute the total number of neighbors

- start \( k \) threads, one for each element of \( Q_{BFS} \)

- build \( Q_{deg} \), by substituting each vertex with its degree

- perform a prefix-sum operation on \( Q_{deg} \) to build the New Offset array (by using the Thrust library)
Sort-Unique BFS

Step 1: build the new offset array and compute the total number of neighbors

- start \( k \) threads, one for each element of \( Q_{BFS} \)
- build \( Q_{deg} \), by substituting each vertex with its degree
- perform a prefix-sum operation on \( Q_{deg} \) to build the New Offset array (by using the Thrust library)
Sort-Unique BFS

Step 1: build the new offset array and compute the total number of neighbors

- start \( k \) threads, one for each element of \( Q_{BFS} \)
- build \( Q_{deg} \), by substituting each vertex with its degree
- perform a prefix-sum operation on \( Q_{deg} \) to build the New Offset array (by using the Thrust library)

The last element of New Offset is: 

\[
m = \sum_{i \in Q_{BFS}} d_i
\]
Sort-Unique BFS

Step 2: map threads to neighbors and build a contiguous array of neighbors

- start $m$ threads
- each thread finds of which vertex and edge is in charge
- each thread reads, from the CSR, the other vertex of its edge
- and writes it in the **Next Level Frontier Set** in the position corresponding to its CUDA index.
Sort-Unique BFS

Step 2: map threads to neighbors and build a contiguous array of neighbors

- start $m$ threads
- each thread finds of which vertex and edge is in charge
- each thread reads, from the CSR, the other vertex of its edge
- and writes it in the Next Level Frontier Set in the position corresponding to its CUDA index.
Sort-Unique BFS

Step 2: map threads to neighbors and build a contiguous array of neighbors

- start $m$ threads
- each thread finds of which vertex and edge is in charge
- each thread reads, from the CSR, the other vertex of its edge
- and writes it in the **Next Level Frontier Set** in the position corresponding to its CUDA index.
Sort-Unique BFS

Step 2: map threads to neighbors and build a contiguous array of neighbors

- start \( m \) threads
- each thread finds of which vertex and edge is in charge
- each thread reads, from the CSR, the other vertex of its edge
- and writes it in the **Next Level Frontier Set** in the position corresponding to its CUDA index.
Sort-Unique BFS
Step 3: prune the Next Level Frontier Set

- start $m$ threads
- perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the *Thrust* library)
- and compact it to $n$ unique elements
Sort-Unique BFS

**Step 3: prune the Next Level Frontier Set**

- Start $m$ threads

- Perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the *Thrust* library)

- And compact it to $n$ unique elements
Sort-Unique BFS

Step 3: prune the Next Level Frontier Set

- start \( m \) threads
- perform a sort-unique operation on the Next Level Frontier Set (by using the Thrust library)
- and compact it to \( n \) unique elements
Sort-Unique BFS

Step 3: prune the Next Level Frontier Set

- start \( m \) threads

- perform a **sort-unique** operation on the **Next Level Frontier Set** (by using the *Thrust* library)

- and compact it to \( n \) unique elements

- Unique ratio \( \frac{m}{n} \sim 20 \)
Sort-Unique BFS: communication and enqueue

Step 4: Exchange vertices and update the parent array

- start \( n \) threads

- substitute vertices with tasks

- sort by tasks (by using the *Thrust* library)

- exchange non-local edges

- update the parent array and enqueue

- If \( Q_{BFS} == 0 \) end.
Sort-Unique BFS: Results

Weak scaling plot, Kernel 2

(infiniband cluster @ Cineca, 32 nodes: 2xM2070, 2x six-core intel Westmere)

On 1 GPU $2^{21}$ vertices, on 64 GPUs $2^{27}$ vertices
Running time breakdown

Breakdown of running time, proc 0 of 32

Cuda Kernels
- binary search
- sort-unique
- cudaMemcpy
- mpi allgather
- mpi send/recv
- mpi allreduce
- mpi recv

Communications
- mpi allgather
- mpi send/recv
- mpi allreduce

CudaCpy
- cudaMemcpy
- cudaMemcpy
- cudaMemcpy
- cudaMemcpy

BFS on GPU clusters
Sort-Unique BFS: Timing

- Time spent in computation is almost constant
- Time spent in communication increases

<table>
<thead>
<tr>
<th>SCALE</th>
<th>N</th>
<th>kernels time</th>
<th>mpi time</th>
<th>NLFS</th>
<th>NLFS-after-SU</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>0.68</td>
<td>0.0</td>
<td>38M</td>
<td>1.0M</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>0.85</td>
<td>0.1</td>
<td>38M</td>
<td>1.7M</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>0.85</td>
<td>0.4</td>
<td>38M</td>
<td>2.7M</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>0.85</td>
<td>0.5</td>
<td>58M</td>
<td>4.5M</td>
</tr>
<tr>
<td>25</td>
<td>16</td>
<td>0.9</td>
<td>0.6</td>
<td>45M</td>
<td>5.5M</td>
</tr>
<tr>
<td>26</td>
<td>32</td>
<td>0.95</td>
<td>0.7</td>
<td>59M</td>
<td>8.7M</td>
</tr>
<tr>
<td>27</td>
<td>64</td>
<td>1.01</td>
<td>0.9</td>
<td>42M</td>
<td>9.3M</td>
</tr>
</tbody>
</table>
Reducing the communication burden

- There are at most $2^{40}$ vertices (the Graph500 defines 40 as the largest scale)
- A vertex is represented by a 64 bits label
- In a distributed implementation no more than 32 bits are required to label local vertices
- There is no need to exchange vertices by using their global label. By exchanging vertices in local format halves the communication.
Reducing the communication burden

Mean MPI time (sec)
- 64 bits: 0.1359
- 32 bits: 0.1131

TEPS
- 64 bits: 7.39E+08
- 32 bits: 9.85E+08

32 MPI tasks. Scale 26.
CSR Relabel

Build a bitmask of owned vertices to reduce the number of duplications.

- start \( m \) threads
- sort the CSR
- relabel vertices from 0 to \( V_{\text{MAX}} - 1 \). The maximum label is also the number of unique vertices in the CSR
- build a bitmask with \( V_{\text{MAX}} \) elements
- sort back the relabelled CSR to build an array of pointers to the bitmask (by using the original CUDA index).

The size of this *local* bitmask is smaller, with respect to a global one. For a graph with \( 2^{25} \) vertices, the CSR has \( 2^{28} \) elements, the bitmask has only \( 2^{22} \) elements.

**Done once! Used before the pruning step.**
What about Kepler?

- Very preliminary results!
- No special tuning, just re-compilation with `arch=sm_35`

<table>
<thead>
<tr>
<th>SCALE</th>
<th>N</th>
<th>TEPS on Fermi</th>
<th>TEPS on Kepler</th>
<th>TEPS on Kepler by using bitmask</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>1</td>
<td>5.22e+07</td>
<td>8.19e+07</td>
<td>2.20e+08</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>9.08e+07</td>
<td>1.46e+08</td>
<td>3.36e+08</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
<td>1.70e+08</td>
<td>2.60e+08</td>
<td>5.31e+08</td>
</tr>
<tr>
<td>24</td>
<td>8</td>
<td>3.13e+08</td>
<td>4.56e+08</td>
<td>7.98e+08</td>
</tr>
<tr>
<td>25</td>
<td>16</td>
<td>5.62e+08</td>
<td>7.30e+08</td>
<td>1.13e+09</td>
</tr>
</tbody>
</table>
Breadth First Search on APEnet+

Data exchange between Device and Host:

- (In general) GPUs cannot exchange data directly
- CPU-GPU copies represent an issue
- recent GPUs support the *peer-to-peer* mechanism

![Diagram showing data exchange between CPU and GPU before and after CUDA 4.0](image)

- The APEnet hardware allows for a direct communication between two GPUs
- All point-to-point MPI communications are replaced by primitives that use the RDMA features of APEnet.
Results: BFS on APEnet+

- OpenMPI/ QDR IB using MPI_Send/Recv
- APEnet+ using native RDMA PUT (needs padding)

Traversed Edges Per Second, Strong Scaling, $|V| = 2^{20}$

<table>
<thead>
<tr>
<th>NP</th>
<th>INFINIBAND</th>
<th>APENET</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$6.2 \times 10^7$</td>
<td>$6.2 \times 10^7$</td>
</tr>
<tr>
<td>2</td>
<td>$7.8 \times 10^7$</td>
<td>$1.0 \times 10^8$</td>
</tr>
<tr>
<td>4</td>
<td>$8.2 \times 10^7$</td>
<td>$1.3 \times 10^8$</td>
</tr>
</tbody>
</table>

Traversed Edges Per Second, Weak Scaling, $|V| = 2^{SCALE}$

<table>
<thead>
<tr>
<th>NP</th>
<th>SCALE</th>
<th>MPI/IB</th>
<th>APEnet+</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
<td>$5.6 \times 10^7$</td>
<td>$6.0 \times 10^7$</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>$7.9 \times 10^7$</td>
<td>$1.0 \times 10^8$</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td>$1.1 \times 10^8$</td>
<td>$1.5 \times 10^8$</td>
</tr>
</tbody>
</table>
Breakdown of running times, $|V|=2^{20}$

Figure: Breakdown of the execution time for one out of four tasks for both APEnet and Infiniband.
## Conclusions

### Summary

- First distributed BFS on a multi-GPU system
- The technique to map threads to data fully exploits the GPU power and may be applied to other problems
- Good scaling properties
- Billions of TEPS with a small cluster of GPUs

### Future Work

- Overlap GPU computation with communication
- Explore different graph partitioning techniques.
Thank You!